# PRINGL Example – Rotating Presidency

Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
{oscekic,truong,dustdar}@dsg.tuwien.ac.at
http://dsg.tuwien.ac.at

## 1 Use Case

A company wants to set up a *rotating presidency* incentive scheme, in which the team becomes managed in the upcoming iteration by the currently best-performing team member, unless that team member was already presiding over the team in the past $k$ iterations. The scheme motivates the best workers psychologically by offering them a more prestigious position in the hierarchy. However, in order to keep team connectedness in a longer run, foster equality and fresh leadership ideas, a single person is prevented from staying too long in the managerial position, and is replaced by the second-best team member (Figure 1).

This example was chosen to additionally showcase the basic use of *structural incentives*. Structural incentives are the incentives that consider relationships between workers (e.g., past collaborations, managerial relations, friendships) within: a) incentive application conditions (filters); and/or b) rewarding actions, by 're-chaining' the relations in order to accomplish a goal (e.g., establish new collaborations, alter communication links, promote, demote, change data flows).

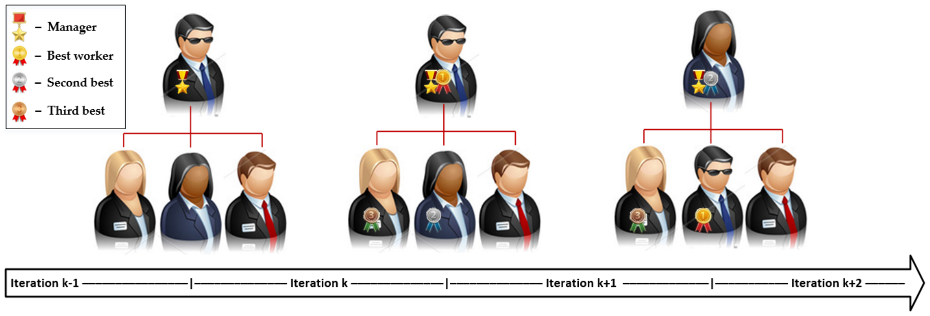


**Fig. 1.** Rotating presidency incentive scheme with maximum two consecutive terms.

## 2 Implementation with PRINGL

Differently than in other examples, here we present the solution in a top-down approach, as it is more understandable for a reader already acquainted with PRINGL's features.

### 2.1 Incentive Scheme

In Figure 2 an incentive scheme named `RotatingPresidency` is defined. It contains a set of global parameters that are used for configuring the execution of the scheme[1]. The integer `teamID` uniquely defines the team that we want the scheme applied to. The `iters` parameter specifies the maximum number of consecutive iterations a team member is allowed to spend as a manager.

The scheme consists of two $\boxed{\texttt{IM}}$s with the same priority. This means that the order of execution is unknown, implying that the two mechanisms should not depend on each other's outcomes. As we will see, in this case we accomplish this by specifying mutually exclusive incentive conditions. The `RewardBest` mechanism installs the best worker as the new manager if (s)he is not the manager already. The `PreventTooLong` mechanism will replace the current manager if (s)he stayed too long in the position, even if the manager resulted again as the best performing team member. 'Installing' or 'replacing' a manager is actually performed by re-chaining of management relations in the structural model of the team in the abstraction interlayer.

### 2.2 Incentive Mechanisms

The two $\boxed{\texttt{IM}}$s (`RewardBest` and `PreventTooLong`) necessary for modeling the rotating presidency scheme are also shown in Figure 2. Both $\boxed{\texttt{IM}}$s get executed always as the nullified `exec_cond` fields default to `true`. The associated `filter`s are passed the collection of all workers. As explained in Section 2.4, the F▷ `Candidates` will return potential candidates for the manager position – the best performing `Worker` and the current manager. In case it is the same person, the returned collection will contain a single `Worker` element.

As $\boxed{\texttt{IM}}$ `RewardBest`'s `inc_cond` field is non-null PRINGL automatically passes the output of the `filter` to the `inc_cond` ◇P element for further evaluation[2]. In this case, this means that the manager candidates are passed to the ◇P `NotSame` to decide whether another worker performed better in the meantime and the actual manager should be replaced. If that is the case (`inc_cond` evaluates to `true`) the `rew_action` /A\ `RewBest` is triggered and passed the candidates.

Similarly, the $\boxed{\texttt{IM}}$ `PreventTooLong` invokes the same filter. However, it has a complex incentive condition field, referring to two ◇P elements, which both need be visually declared. PRINGL allows this as a shorthand notation instead of forcing the user to create a container ◇P element to perform the same logical function. In this case, the exposed parameters cannot be simply referenced by using the field name, but rather the parameters are accessed through an associative array (C# Dictionary) bearing the same name as the field, while the names of the

[1] For the purposes of this demonstration, we assume that the iterations and performance metrics are defined and managed at the abstraction interlayer (RMod). This scheme uses the 'effort' metric, but any other compatible performance metric could have been used in this demo. We also assume the abstraction interlayer triggers the scheme execution upon each relevant iteration's end.

[2] PRINGL passes also the other auto-parameters specified in Table 3 in the original paper, but we do not list them here if they are not relevant to this example.

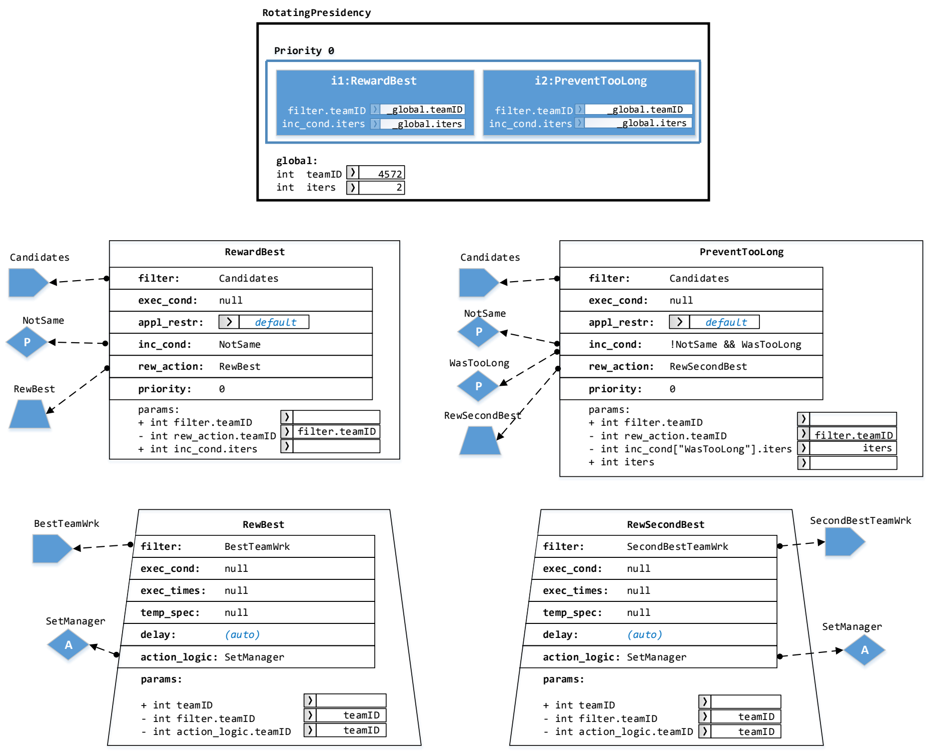


**Fig. 2.** Modeling the rotating presidency incentive strategy in PRINGL. Segment showing the scheme, mechanisms and rewarding actions.

used ⟨P⟩ elements serve as key names. For example, to access the ⟨P⟩ `WasTooLong`'s parameter `iters` from $\boxed{\text{IM}}$ `PreventTooLong` where ⟨P⟩ `WasTooLong` is used in the `inc_cond` field, we must write: `inc_cond["WasTooLong"].iters` As it can be visually tiring to read the lengthy fully-qualified names of propagated parameters, we often stop propagating such parameters and propagate a new, local one with the same name, whose value we then copy to the long-named parameter (e.g., just `iters` instead of `inc_cond["WasTooLong"].iters`).

The $\boxed{\text{IM}}$ `PreventTooLong`'s full incentive condition is: `!NotSame && WasTooLong`. The first part of this condition ensures that the rewarding action /A\ `RewSecondBest` of $\boxed{\text{IM}}$ `PreventTooLong` will never get executed at the same time as the /A\ `RewBest` of the $\boxed{\text{IM}}$ `RewardBest`.[3]

If the composite incentive condition evaluates to `true`, this means that the actual manager occupied the position for too long, and that it should be now

[3] By scheduling the rewarding actions in its *Timeline* mechanism the abstraction interlayer (RMod) takes care that all the filters read the same values and that the order of execution of $\boxed{\text{IM}}$s will not affect those values before the execution of the whole scheme is completed.

replaced by the second-best worker. To do this we invoke the A `RewSecondBest` and pass to it the collection of workers returned by the `filter` F `Candidates`.

## 2.3 Rewarding Actions

The rewarding actions `RewBest` and `RewSecondBest` (Figure 2) are almost identical, differing only in the filter they use – with former using the `BestTeamWrk` and the latter the `SecondBestTeamWrk`. The fact that a rewarding action A declares its own `filter` means that it discards the workers passed to it by the PRINGL environment from the encompassing IM's `filter` and rewards those returned by the local filter. In both actions most fields are nullified, returning the default values. This means that the `action_logic` A `SetManager` will be unconditionally scheduled for immediate execution in the interlayer. Both actions also do the same parameter name shortening we explained before, purely for readability purposes.

## 2.4 Worker Filters

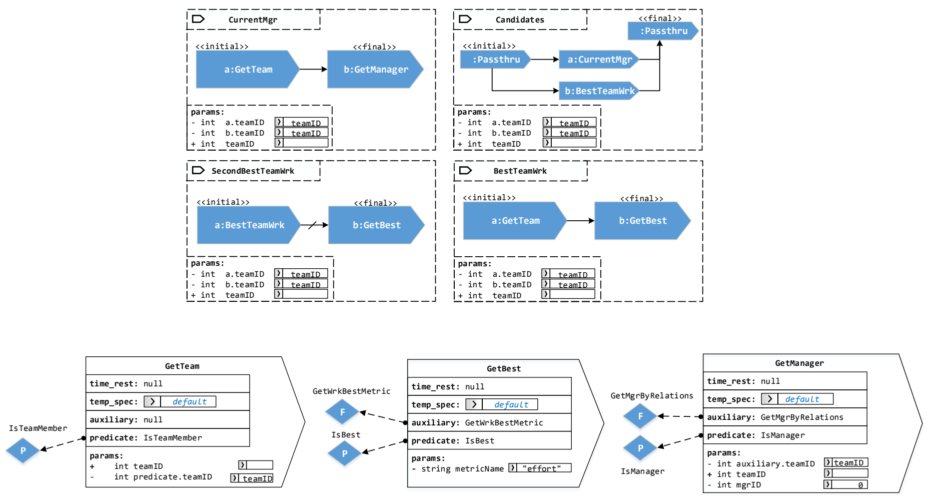


**Fig. 3.** Modeling the rotating presidency incentive strategy in PRINGL. Segment showing the worker filters.

The example requires the definition of four different composite filters, shown in Figure 3:

1. F `CurrentMgr` – Returns the current manager of the team. The F `a:GetTeam` returns all the workers belonging to the team with the `teamID`, while the F `b:GetManager` uses managerial relationships to determine the manager among those workers[4].

[4] While managerial relations in principle need not be stored as a graph, and can thus be identified much more easily, we still use the graph managerial relations as an

2. 𝔽 `BestTeamWrk` – Returns the best individual from a previously identified collection of workers belonging to a team. The 𝔽 `b:GetBest` itself decides what 'best worker' means in this case (see later).

3. 𝔽 `SecondBestTeamWrk` – As the name suggests, returns the second best worker in the team. The filter 𝔽 `a:BestTeamWrk` returns the best worker of the team and passes it forward to the 𝔽 `b:GetBest` via a negated edge ($\nrightarrow$). This means that 𝔽 `b` now receives as input: *input(a)* $\setminus$ 𝔽 `a`, i.e., in this particular case the collection of all workers belonging to the team minus the best worker. Filter `b` returns the best worker from this collection, and thus effectively the second best worker of the team.

4. 𝔽 `Candidates` – This filter simply uses the previously defined filters `CurrentMgr` and `BestTeamWrk` and returns the set union of their resulting collections. In order to parallelize filters and produce a union without performing any other additional filtering, we can use as convenience the anonymous, built-in *pass-through* filters. They contain no logic, except for a predicate always returning `true`. The resulting complex `Candidates` filter can return at most two workers.

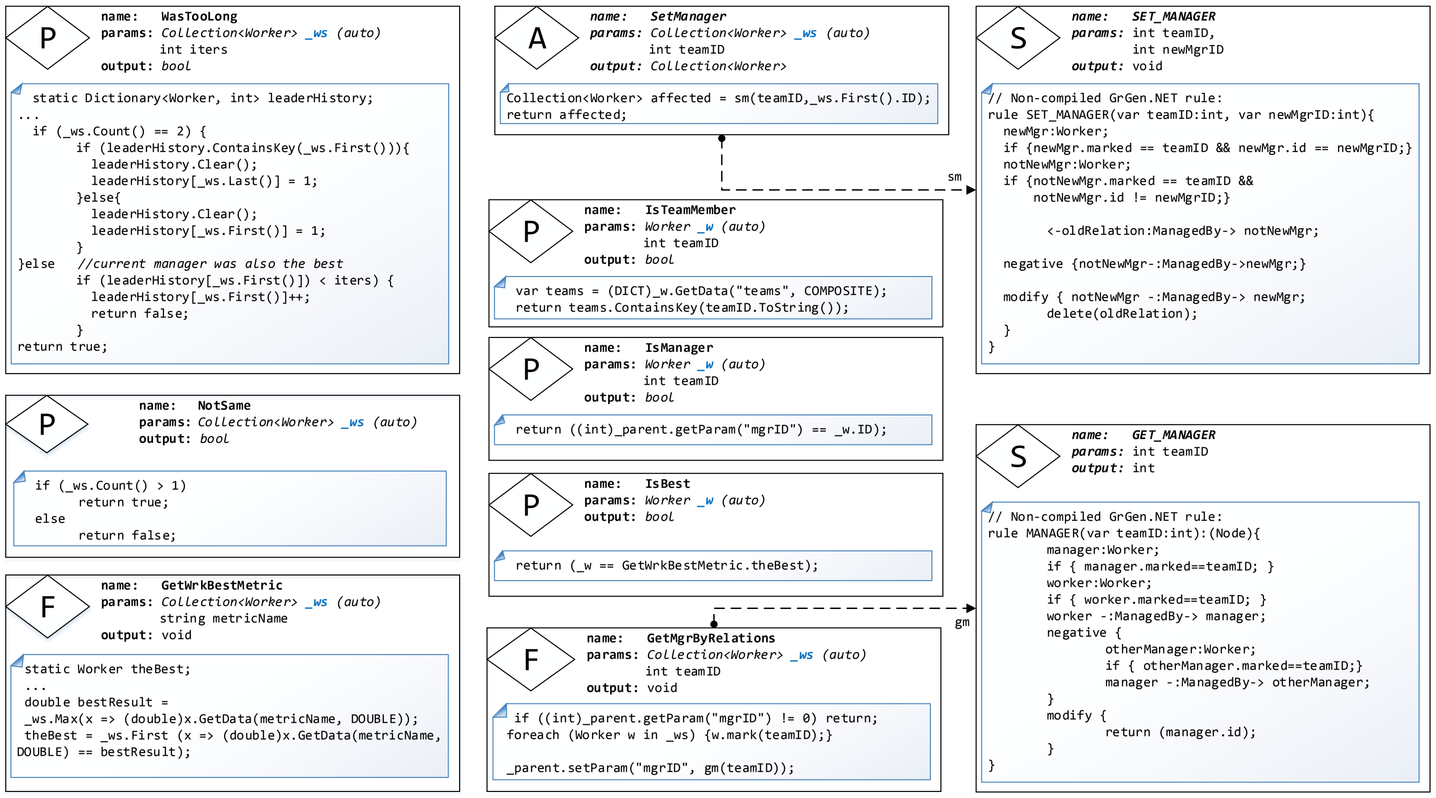
All composite filters require only one parameter (`teamID`) and perform parameter name shortening. They are built by composing these simple filters:

1. 𝔽 `GetTeam` – Returns all the workers being listed as belonging to the team with specified `teamID`. The filtering is performed by running each of the workers from the input set against the `predicate` ⟨ℙ⟩ `IsTeamMember` and including it in the output if fulfilling the predicate.

2. 𝔽 `GetBest` – Returns the worker having achieved the highest value of the 'effort' metric by invoking the ⟨𝔽⟩ `GetWrkBestMetric` and then just formally matching it with the `IsBest` predicate.
   As we can see, in our example this filter encapsulates and hides the metric it uses for evaluating the workers. In principle, it would make sense to propagate the metric name upwards and thus make it user-settable, consequently making the whole scheme more general. However, for readability purposes we decided not to propagate this parameter in this example. It is interesting to notice that this filter does not care to which team the evaluated worker belongs – if used independently, it would try to evaluate all the workers in the system. This is why we always use it in composite filters, where we initially restrict its input set with another filter.

3. 𝔽 `GetManager` – Invokes a ⟨𝔽⟩ `GetMgrByRelations` that performs a graph query on the team model in abstraction interlayer to determine the manager within the provided input set of workers. We explain how it works in more detail in the following Section 2.5.

easily understandable example of how any graph-encoded structural property can be used in incentive management.



```
name:   WasTooLong
params: Collection<Worker> _ws (auto)
        int iters
output: bool

 static Dictionary<Worker, int> leaderHistory;
...
  if (_ws.Count() == 2) {
        if (leaderHistory.ContainsKey(_ws.First())){
          leaderHistory.Clear();
          leaderHistory[_ws.Last()] = 1;
        }else{
          leaderHistory.Clear();
          leaderHistory[_ws.First()] = 1;
        }
}else   //current manager was also the best
        if (leaderHistory[_ws.First()]) < iters) {
          leaderHistory[_ws.First()]++;
          return false;
        }
return true;
```

```
name:   NotSame
params: Collection<Worker> _ws (auto)
output: bool

 if (_ws.Count() > 1)
       return true;
 else
       return false;
```

```
name:   GetWrkBestMetric
params: Collection<Worker> _ws (auto)
        string metricName
output: void

static Worker theBest;
 ...
 double bestResult =
 _ws.Max(x => (double)x.GetData(metricName, DOUBLE));
 theBest = _ws.First (x => (double)x.GetData(metricName,
DOUBLE) == bestResult);
```

```
name:   SetManager
params: Collection<Worker> _ws (auto)
        int teamID
output: Collection<Worker>

Collection<Worker> affected = sm(teamID,_ws.First().ID);
return affected;
```

```
name:   IsTeamMember
params: Worker _w (auto)
        int teamID
output: bool

 var teams = (DICT)_w.GetData("teams", COMPOSITE);
 return teams.ContainsKey(teamID.ToString());
```

```
name:   IsManager
params: Worker _w (auto)
        int teamID
output: bool

 return ((int)_parent.getParam("mgrID") == _w.ID);
```

```
name:   IsBest
params: Worker _w (auto)
output: bool

 return (_w == GetWrkBestMetric.theBest);
```

```
name:   GetMgrByRelations
params: Collection<Worker> _ws (auto)
        int teamID
output: void

 if ((int)_parent.getParam("mgrID") != 0) return;
foreach (Worker w in _ws) {w.mark(teamID);}

_parent.setParam("mgrID", gm(teamID));
```

```
name:   SET_MANAGER
params: int teamID,
        int newMgrID
output: void

// Non-compiled GrGen.NET rule:
rule SET_MANAGER(var teamID:int, var newMgrID:int){
  newMgr:Worker;
  if {newMgr.marked == teamID && newMgr.id == newMgrID;}
  notNewMgr:Worker;
  if {notNewMgr.marked == teamID &&
      notNewMgr.id != newMgrID;}

        <-oldRelation:ManagedBy-> notNewMgr;

  negative {notNewMgr-:ManagedBy->newMgr;}

  modify { notNewMgr -:ManagedBy-> newMgr;
        delete(oldRelation);
  }
}
```

```
name:   GET_MANAGER
params: int teamID
output: int

// Non-compiled GrGen.NET rule:
rule MANAGER(var teamID:int):(Node){
        manager:Worker;
        if { manager.marked==teamID; }
        worker:Worker;
        if { worker.marked==teamID; }
        worker -:ManagedBy-> manager;
        negative {
                otherManager:Worker;
                if { otherManager.marked==teamID;}
                manager -:ManagedBy-> otherManager;
        }
        modify {
                return (manager.id);
        }
}
```

**Fig. 4.** Modeling the rotating presidency incentive strategy in PRINGL. Segment showing the incentive logic.

### 2.5 Incentive Logic

Incentive logic elements contain the low-level business logic and code that communicates directly with the abstraction interlayer. They are shown in Figure 4. We use C# in this example in all but ⟨S⟩ elements, which are shown in the original GrGen.NET rule language[5] that eventually also gets compiled into C# code.

The *(auto)* parameters are those passed by PRINGL transparently to the user, according to the Table 3 in the original paper. All auto-parameters listed in the table for a given incentive logic type are always passed. Due to space constraints, we show only those that are used in this example. This is why a ⟨P⟩ element is sometimes shown with a single `Worker _w` auto-parameter, while in other cases we have a `Collection<Worker> _ws` parameter. In reality, both are always passed, along with other parameters as specified in the table. Apart from the auto-parameters, the incentive designer is free to use a number of arbitrary parameters.

Now we describe the functionality of the incentive elements we use in this example, focusing on the elements whose function is not straightforward.

1. ⟨P⟩ `IsTeamMember` – Determines whether a worker belongs to a team.

2. ⟨P⟩ `IsManager` – Checks if the currently evaluated worker has the ID previously determined to belong to the team manager by the `GetMgrByRelations auxiliary`.

3. ⟨P⟩ `IsBest` – Checks if the currently evaluated worker is the same as the one identified by the `GetWrkBestMetric auxiliary`. Differently from how we do in `IsManager`, here we do not use the parent's local parameter to pass the information, but rather a public static variable `GetWrkBestMetric.theBest`. This can have as a consequence that an incorrect value can be read if `GetWrkBestMetric` is also used for assessing another team at the same time. However, we do it in this way here purely for demonstrative purposes.

4. ⟨P⟩ `NotSame` – Returns true if the input collection contains two manager candidates.

5. ⟨P⟩ `WasTooLong` – Keeps track of how many times a worker was in the manager position, and returns true if the worker is not supposed to become manager in the upcoming iteration. The input collection can contain only 1 or 2 elements, as it gets forwarded from the `Candidates filter` within the `PreventTooLong` $\boxed{\text{IM}}$.

6. ⟨F⟩ `GetWrkBestMetric` – Reads the value of the `'effort'` metric for each of the passed workers in `_ws` from the abstraction interlayer and updates the best worker.

[5] `www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`

7. ⟨F⟩ `GetMgrByRelations` – Prepares the ground for correct execution of the structural queries by marking the nodes of the graph model in the abstraction interlayer corresponding to the team members with the appropriate `teamID` tag. It then invokes the ⟨S⟩ `GET_MANAGER` that contains the compiled graph query matching a graph node that corresponds to the actual manager.

8. ⟨A⟩ `SetManager` – A wrapper for invocation of the ⟨S⟩ `SET_MANAGER`.

9. ⟨S⟩ `GET_MANAGER` – Contains a compiled non-modifying GrGen.NET graph query, here expressed in their proprietary rule language. The rule only considers the nodes marked by the `teamID` tag (see `GetMgrByRelations`). The rule matches and returns a node that other nodes point to via `ManagedBy`-typed relations, and that itself is not managed by another team member.

10. ⟨S⟩ `SET_MANAGER` – Contains a compiled modifying GrGen.NET graph query, here expressed in their proprietary rule language. The rule only considers the nodes marked by the `teamID` tag (see `GetMgrByRelations`). It matches the old and the new manager, and re-chains the `ManagedBy` relations to point to the new manager node.

## 3 Discussion

### 3.1 Composability, Reusability and Ease of Use

Filters like `GetTeam`, `GetBest` and `GetManager` perform very common incentive functionality (business logic). In practice, this means that we can expect to have such components available as library elements. Of course, if we need to use a company-specific `auxiliary` or `predicate`, we can easily replace the default one with a proprietary element. For example, a $\boxed{F}$ `GetManager` may be available with a default `auxiliary` ⟨F⟩ that looks for a manager in the team model by inspecting the node tags for a given manager tag. In that case, we may want to exchange the default, tag-based ⟨F⟩ with a structural one, such as `GetMgrByRelations`. Once we have simple filters available, composing more complex ones can be done in a matter of minutes by purely visual modelling.

The two rewarding actions we employ here are very similar. Once we model one of them, the other one can be created by copy-pasting and connecting a different filter. The `action_logic` ⟨A⟩ for installing a new manager is also a common piece of incentive logic that can be implemented in different flavors and provided as a library element.

All these elements can be put to use in $\boxed{IM}$ definitions, where we can easily interchange different filters and rewarding actions, or tweak parameters to obtain specific incentive mechanisms we want. For example, the /A\ `RewBest` can use a completely different `action_logic` ⟨A⟩ to reward the best workers – for example to pay out money instead of changing team managers. This allows partial adaptations of the incentive strategy.

The same principle can be applied at the incentive scheme level. For example, a company that applies the rewarding strategy of always awarding the team leadership to the best performing worker would only need the $\boxed{\texttt{IM}}$ `RewardBest` in the scheme. If later it decided to switch to the rotating presidency (i.e., to limit the number of consecutive terms) it would just need to augment the old scheme with another $\boxed{\texttt{IM}}$ – `PreventTooLong`. It is easy to imagine how other constraints could be added.

### 3.2 Structural Incentives

Incentive schemes in use today usually neglect the relationships between workers, both when assessing workers, and when awarding them[1]. However, a worker's performance can often be better understood if considered in the context of his surroundings and collaborations (collaborators). At the same time, measures like:

- placing a worker to work with different co-workers
- offering a different place in the organizational hierarchy
- introduction of new collaborative patterns
- introduction communication methods

all can have a significant motivational impact on the worker[1]. This is why PRINGL supports formally specifying such *structural incentive measures*. In [2] we presented the rewarding framework PRINC whose component RMod provides the abstraction interlayer functionality for PRINGL. RMod models the workers and their relations as a multi-graph. The graph model was chosen as the most generic one, yielding an opportunity to map it into different company-specific models. At the same time, this graph model gives us the opportunity to explore modeling structural incentives. We use a well-known tool (GrGen) and established techniques for graph rewriting as the basis for our structural incentives. In this example we showed a small example of how they can be performed. They will be in the focus of our future work.

## References

1. Scekic, O., Truong, H.L., Dustdar, S.: Incentives and rewarding in social computing. Communications of the ACM **56**(6) (June 2013) 72–82
2. Scekic, O., Truong, H., Dustdar, S.: Programming incentives in information systems. In: Int. Conf. on Advanced Inf. Sys. Engineering CAiSE'13. (2013) 688–703